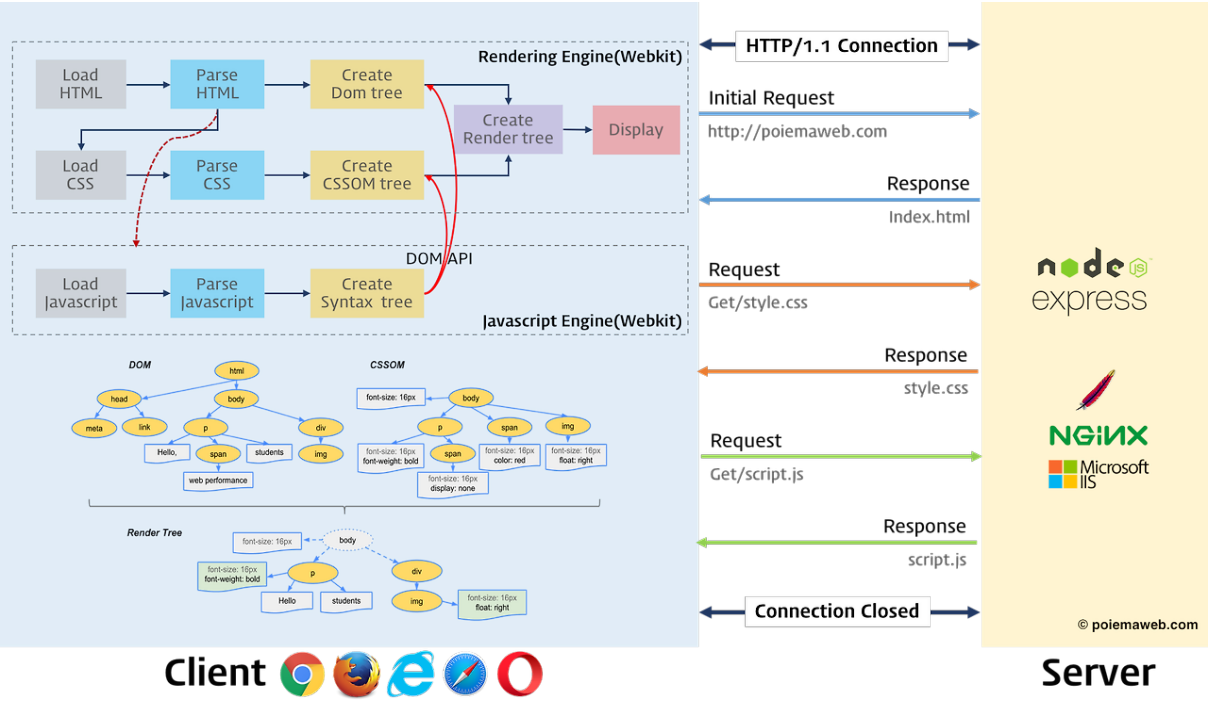
38장. 브라우저의 렌더링 과정 ⭐

1. 브라우저가 HTMl, CSS, JS로 작성된 텍스트 문서를 어떻게 파싱(해석)하여 브라우저에 렌더링하는지 살펴보자.

- 파싱(parsing)
- 렌더링(rendering)



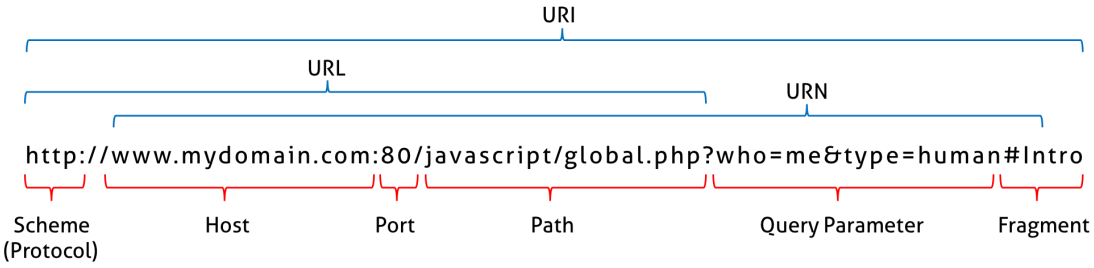
브라우저의 렌더링 과정을 간략하게 표현한 것

2. 브라우저는 다음과 같은 과정을 거쳐 렌더링을 수행한다.
    a. 브라우저는 HTML, CSS, JS, 이미지, 폰트 파일 등 렌더링에 필요한 리소스를 요청하고 서버로부터 응답을 받는다.
    b. 브라우저의 렌더링 엔진은 서버로부터 응단된 HTML, CSS를 파싱하여 DOM과 CSSOM을 생성하고 이들을 결합하여 렌더 트리를 생성한다.
    c. 브라우저의 자바스크립트 엔진은 서버로부터 응답된 자바스크립트를 파싱하여 AST를 생성하고 바이트코드로 변환하여 실행한다. 이때 자바스크립트는 DOM API를 통해 DOM이나 CSSOM을 변경할 수 있다. 변경된 DOM과 CSSOM은 다시 렌더 트리로 결합된다.
    d. 렌터 트리를 기반으로 HTML 요소의 레이아웃(위치와 크기)을 계산하고 브라우저 화면에 HTML 요소를 페인팅한다.

38.1 요청과 응답

1. **브라우저의 핵심 기능**은 필요한 리소스(HTML, CSS, JS, 이미지, 파일 등)를 서버에 요청하고 서버로부터응답 받아 브라우저에 시각적으로 렌더링하는 것.

2. 서버에 요청을 하기 위해서는 브라우저 주소창에 URL을 입력하고 엔터키를 누른다.URL의 호스트 이름이 DNS를 통해 IP 주소로 변환되고 이 IP 주소를 갖는 서버에게 요청을 전송한다.



3. 예를들어 브라우저에 "https://poiemaweb.com" 을 입력하면 DNS 서버에서 주소에 해당하는 IP를 반환해 주고 접속하게 된다. 그 후 서버에서 주소에 해당하는 파일들을 내려준다.

4. 브라우저의 렌더링 엔진은 받은 HTML(index.html)을 파싱하는 도중에 외부 리소스를 로드하는 태그, 즉 CSS 파일을 로드하는 link 태그, 이미지 파일을 로드하는 img 태그, JS를 로드하는 script 태그 등을 만나면 **HTML의 파싱을 일시 중단하고 해당 리소스 파일을 서버로 요청**한다.

38.2 HTTP 1.1과 HTTP 2.0

1. HTTP/1.1은 기본적으로 커넥션당 하나의 요청과 응답만 처리.

   → 요청할 리소스 개수에 비례하여 응답 시간이 증가하는 단점

2. HTTP/2는 커넥션당 여러 개의 요청과 응답, 즉 다중 요청/응답이 가능.

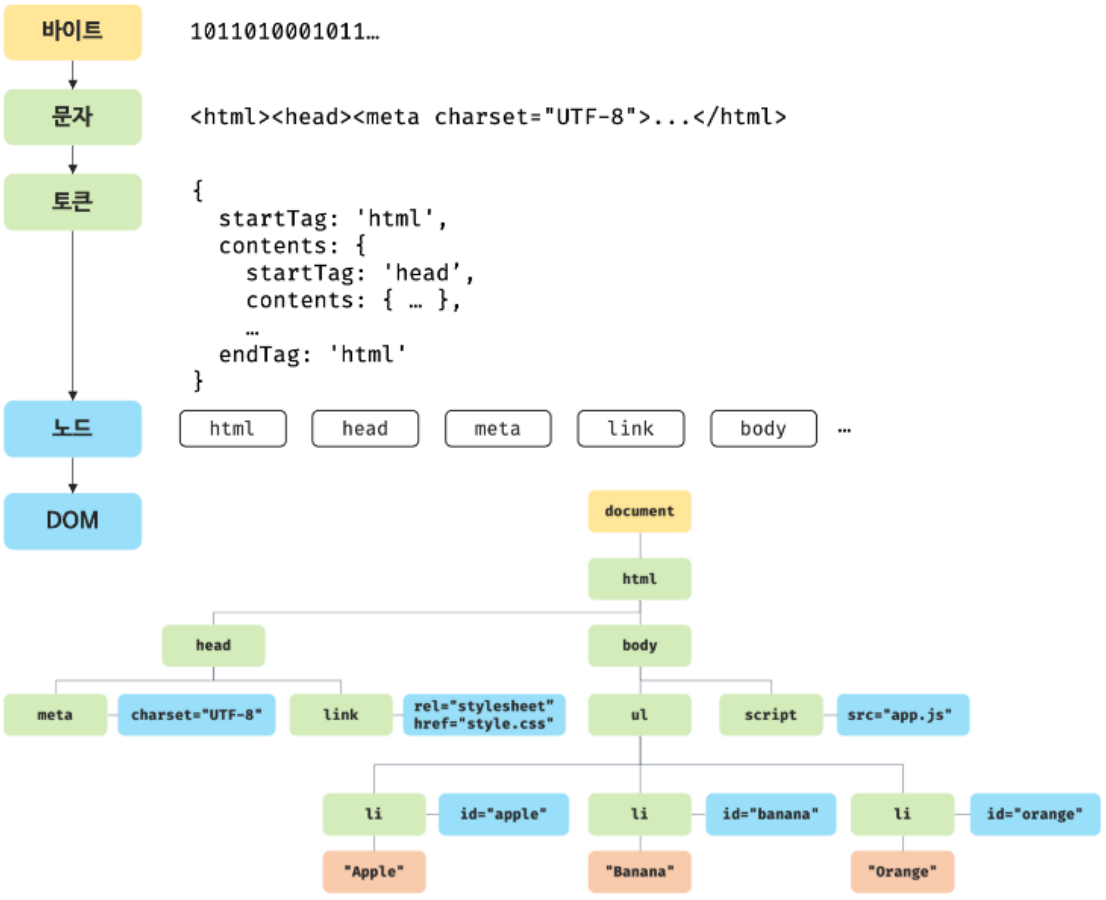
   → HTTP/1.1에 비해 페이지 로드 속도가 **약 50%** 정도 빠르다고 알려짐.

38.3 HTML 파싱과 DOM 생성

1. 브라우저의 요청에 의해 서버가 응답한 HTML 문서는 문자열로 이루어진 순수한 텍스트.

   순수한 텍스트인 HTML 문서를 브라우저에 시각적인 픽셀로 렌더링하려면,

   HTML 문서를 브라우저가 이해할 수 있는 자료구조(객체)로 변환하여 메모리에 저장해야 함.



1. 서버에 존재하던 HTML 파일이 브라우저의 요청에 의해 응답된다. 이때 서버는 브라우저가 요청한 HTML 파일을 읽어 들여 메모리에 저장한 다음 메모리에 저장된 바이트(2진수)를 인터넷을 경유하여 응답한다.
2. 브라우저는 서버가 응답한 HTML 문서를 바이트(2진수) 형태로 응답받는다. 그리고 선언된 인코딩 방식(예: UTF-8)을 기준으로 문자열로 변환된다.
3. 문자열로 변환된 HTML 문서를 읽어 들여 문법적 의미를 갖는 코드의 최소 단위인 **토큰**들로 분해한다.
4. 각 토큰들을 객체로 변환하여 **노드**들을 생성한다. 노드는 DOM을 구성하는 기본 요소가 된다.
5. HTML 문서는 HTML 요소들의 집합으로 이루어지며 HTML 요소는 중첩 관계를 갖는다. 중첩 관계에 의해부자 관계가 형성된다. 이러한 부자 관계를 반영하여 모든 노드들을 **트리 자료구조**로 구성한다. 이 노드들로 구성된 트리 자료구조를 **DOM**이라 부른다.

즉, DOM은 HTML 문서를 파싱한 결과물이다.

38.4 CSS 파싱과 CSSOM 생성

1. 렌더링 엔진은 HTML을 처음부터 한 줄씩 순차적으로 파싱하여 DOM을 생성해 나간다.

2. 렌더링 엔진은 DOM을 생성해 나가다가 CSS를 로드하는 link 태그나 style 태그를 만나면 DOM 생성을 일시 중단한다.

3. 그리고 CSS를 HTML과 동일한 파싱 과정 (바이트->문자->토큰->노드->CSSOM)을 거치며 해석하여 CSSOM을 생성한다.

4. CSS 파싱을 완료하면 HTML 파싱이 중단된 지점부터 다시 HTML 파싱을 시작한다.

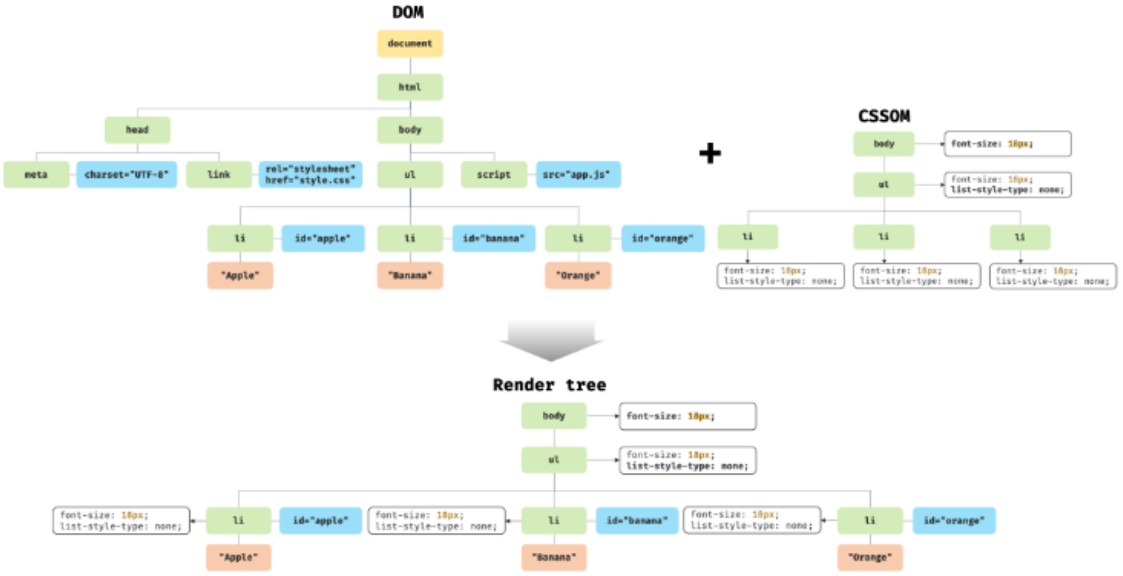
38.5 렌더 트리 생성

1. 렌더링 엔진은 서버로부터 응답된 HTML과 CSS를 파싱하여 각각 DOM과 CSSOM를 생성한다.

2. 그리고 DOM과 CSSOM은 렌더링을 위해 렌더 트리로 결합된다.

   렌더 트리는 렌더링을 위한 트리 구조의 자료구조다. 따라서 브라우저 화면에 렌더링되지 않는노드(meta 태크, script 태그 등)와 CSS에 의해 비표시(display:none)되는 노드들은 포함 X

   다시 말해, **렌더 트리는 브라우저 화면에 렌더링되는 노드만으로 구성된다.**



3. 이후 완성된 렌더 트리는 각 HTML 요소의 레이아웃(위치와 크기)를 계산하는 데 사용되며 브라우저 화면에 픽셀을 렌더링하는 페인팅 처리에 입력된다.

4. 지금까지 살펴본 브라우저 렌더링 과정은 반복해서 실행될 수 있다. 다음과 같은 경우 반복해서레이아웃 계산과 페인팅이 재차 실행된다.
    - 자바스크립트에 의한 노드 추가 또는 삭제
    - 브라우저 창의 리사이징에 의한 뷰포트 크기 변경
    - HTML 요소의 레이아웃 변경을 발생시키는 width/height, margin, padding 등의 스타일 변경

5. 레이아웃 계산과 페인팅을 다시 실행하는 리렌더링은 비용이 많이 들고 성능에 악영향을 주는 작업. 따라서 가급적 리렌더링이 빈번하게 발생하지 않도록 주의할 필요가 있다.

38.6 자바스크립트 파싱과 실행

1. HTML 문서를 파싱한 결과물로 생성된 DOM은 HTML 문서의 구조와 정보뿐만 아니라 HTML 요소와 스타일 등을 변경할 수 있는 프로그래밍 인터페이스로서 DOM API를 제공한다. 자바스크립트 코드에서 DOM API를 사용하면 DOM을 동적으로 조작할 수 있다.

2. CSS 파싱 과정과 마찬가지로 렌더링 엔진은 HTML을 한 줄씩 순차적으로 파싱하여 DOM을 생성해 나가다가 자바스크립트 파일을 로드하는 script 태그나 자바스크립트 코드를 만나면 DOM 생성을 일시 중단한다.

3. 그리고 자바스크립트 코드를 파싱하기 위해 자바스크립트 엔진에 제어권을 넘긴다.이후 자바스크립트 파싱이 종료되면 HTML 파싱이 중단된 지점부터 다시 HTML을 파싱한다.

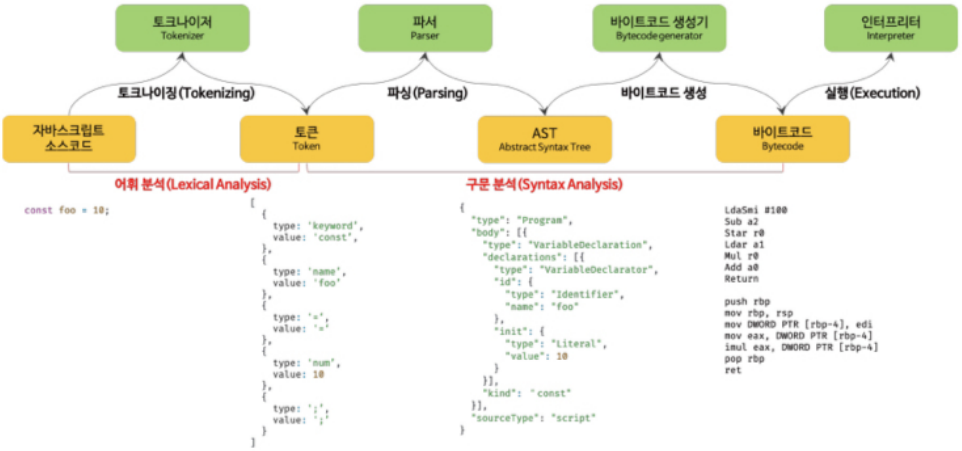


그림 38-10 자바스크립트 파싱과 실행

- 토크나이징(tokenizing)
- 파싱(parsing)

38.7 리플로우와 리페인트

1. 만약 자바스크립트 코드에 DOM이나 CCSOM을 변경하는 DOM API가 사용된 경우 DOM이나 CSSOM이 변경된다.

   이때 변경된 DOM과 CSSOM은다시 렌더 트리로 결합되고 변경된 렌더 트리를 기반으로레이아웃과 페인트 과정을 거쳐 브라우저의 화면에 다시 렌더링한다.

   이를 리플로우(reflow), 리페인트(repaint)라 한다.

2. 리플로우는 레이아웃 계산을 다시 하는 것을 말하며, 노드 추가/삭제, 요소 크기/위치 변경 등 레이아웃에 영향을 주는 변경이 발생한 경우에 한하여 실행된다.

   리페인트는 재결합된 렌더 트리를 기반으로 다시 페인트를 하는 것을 말한다.

38.8 자바스크립트 파싱에 의한 HTML 파싱 중단

1. 지금까지 살펴본 렌더링 엔진과 자바스크립트 엔진은 병렬적으로 파싱을 실행하지 않고 **직렬적으로 파싱을 수행한다**. 즉, HTML문서에서 위에서 아래 방향으로 순차적으로 HTML, CSS, JS를 파싱하고 실행한다.

   이때 script 태그의 위치에 따라 HTML 파싱이 블로킹되어 DOM 생성이 지연될 수 있다. 따라서 script 태그의 위치는 **중요한 의미**를 갖는다.

   만약 자바스크립트 코드가 DOM을 변경하는 DOM API를 사용할 때 DOM의 생성이 완료되지 않은 상태라면 문제가 발생할 수 있다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="style.css">
    <script>
      /*
      DOM API인 document.getElementById는 DOM에서 id가 'apple'인 HTML 요소를
      취득한다. 아래 DOM API가 실행되는 시점에는 아직 id가 'apple'인 HTML 요소를 파싱
      않았기 때문에 DOM에는 id가 'apple'인 HTML 요소가 포함되어 있지 않다.
      따라서 아래 코드는 정상적으로 id가 'apple'인 HTML 요소를 취득하지 못한다.
      */
      const $apple = document.getElementById('apple');

      // id가 'apple'인 HTML 요소의 css color 프로퍼티 값을 변경한다.
      // 이때 DOM에는 id가 'apple'인 HTML 요소가 포함되어 있지 않기 때문에 에러가 발
      $apple.style.color = 'red'; // TypeError: Cannot read property 'styl
    </script>
  </head>
  <body>
    <ul>
      <li id="apple">Apple</li>
      <li id="banana">Banana</li>
      <li id="orange">Orange</li>
    </ul>
```

```
  </body>
</html>
```

2. 이러한 문제를 회피하기 위해 body 요소의 가장 아래에 자바스크립트를 위치시키는 것은 **좋은 아이디어**다.
   - 페이지 로딩 시간이 단축되는 이점도 있다.

38.9 script 태그의 async/defer 어트리뷰트

1. 앞에서 자바스킵트 파싱에 의한 DOM 생성이 중단되는 문제를 근본적으로 해결하기 위해 HTML5부터 script 태그에 async와 defer 어트리뷰트가 추가되었다.

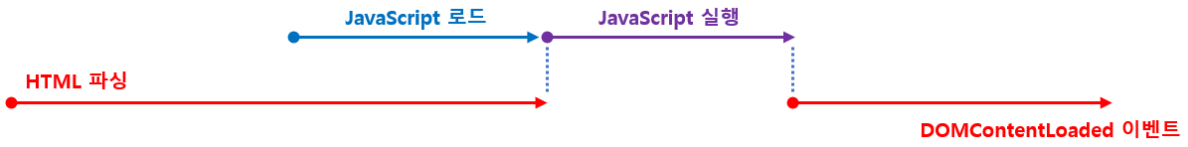
   async와 defer 어트리뷰트는 다음과 같이 src 어트리뷰트를 통해 외부 자바스크립트 파일을 로드하는 경우에만 사용할 수 있다.

   즉, src어트리뷰트가 없는 인라인 자바스크립트에는 사용할 수 없다.

```
<script async src="extern.js"></script>
<script defer src="extern.js"></script>
```

📌 async 어트리뷰트

HTML 파싱과 외부 자바스크립트 파일의 로드가 비동기적으로 동시에 진행된다. 단, 자바스크립트 파싱과 실행은 자바스크립트 파일의 로드가 완료된 직후 진행되며, 이때 HTML 파싱이 중단된다.



여러 개의 script 태그에 async를 지정하면 script 태그의 순서와는 상관없이 로드가 완료된 자바스크립트부터 먼저 실행되므로 순서가 보장되지 않는다.

따라서 순서 보장이 필요한 곳에서는 사용하지 않아야 한다.

📌 defer 어트리뷰트

async와 마찬가지로 HTML 파싱과 외부 자바스크립트 파일의 로드가 비동기적으로 동시에 진행된다. 단, 자바스크립트의 파싱과 실행은 HTML 파싱이 완료된 직후, 즉 DOM 생성이 완료된 직후 진행된다.

따라서 DOM 생성이 완료된 이후 실행되어야 할 자바스크립트에 유용하다.


JavaScript 로드
JavaScript 실행
HTML 파싱
DOMContentLoaded 이벤트